LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-632265

Automatic Problem Localization in Distributed Applications via Multi-dimensional Metric Profiling

I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, A. Gheith

April 3, 2013

32nd International Symposium on Reliable Distributed Systems
Braga, Portugal
September 30, 2013 through October 3, 2013

Disclaimer

Automatic Problem Localization via Multi-dimensional Metric Profiling

Ignacio Laguna[1], Subrata Mitra[2], Fahad A. Arshad[2], Nawanol Theera-Ampornpunt[2], Zongyang Zhu[2], Saurabh Bagchi[2], Samuel P. Midkiff[2], Mike Kistler[3], Ahmed Gheith[3]

[1]*Lawrence Livermore National Laboratory* [2]*Purdue University* [3]*IBM Research Austin*
ilaguna@llnl.gov, {mitra4, faarshad, ntheeraa, zhu85, sbagchi, smidkiff}@purdue.edu, {mkistler, ahmedg}@us.ibm.com

***Abstract*—Debugging today's large-scale distributed applications is complex. Traditional debugging techniques such as breakpoint-based debugging and performance profiling require a substantial amount of domain knowledge and do not automate the process of locating bugs and performance anomalies. We present ORION, a framework to automate the problem-localization process in distributed applications. From a large set of metrics, ORION intelligently chooses important metrics and models the application's runtime behavior through pairwise correlations of those metrics in the system, within multiple non-overlapping time windows. When correlations deviate from those of a learned correct model due to a bug, our analysis pinpoints the metrics and code regions (class and method within it) that are most likely associated with the failure. We demonstrate our framework with several real-world failure cases in distributed applications such as: HBase, Hadoop DFS, a campus-wide Java application, and a regression testing framework from IBM. Our results show that ORION is able to pinpoint the metrics and code regions that developers need to concentrate on to fix the failures.**

***Keywords*-debugging aids; tracing; diagnostics; performance metrics**

I. INTRODUCTION

Debugging today's large-scale distributed systems is complex. Systems are composed of multiple software components often running on distributed nodes. The interactions between these components are complex enough that they cannot all be enumerated a priori. The unpredictability of the execution environment and its effects on the application execution increases difficulty in the debugging process. Failures can come from different layers of the system—network, hardware, operating system, middleware, and application layers. Thus in general, it is necessary to monitor the behavior of all the layers to understand the origin of failures.

Why another debugging tool? There exists a significant number of debugging tools today [1], [2], [3]. They work well for many kinds of failures though they require varying amounts of developer intervention. Despite the existence of this rich set of tools, the debugging process is time consuming and it often requires full domain knowledge of the program to find where problems originate in the source code. Profilers [4], [5], [6] can help in isolating performance bottlenecks, however, identifying the root cause of the bottleneck still remains essentially a manual process. Research on distributed systems has developed functional techniques that can help in debugging, such as program tracing and replay debugging [7], [8], [9], model checking [10], [11], [12], and log analysis [13], [14], [15], [16]. But there remains work to be done to build on these techniques to create a usable debugging tool.

In this work, we focus on debugging distributed applications by identifying the region of code where a fault first becomes active. The developer can then focus on this region to fix the problem rather than spending time in examining the entire source code. We focus on *manifested-on-metrics* (MM) bugs, i.e., those bugs that manifest themselves as an abnormal temporal pattern in one or more metrics at the hardware, OS, middleware, or application layers. MM bugs can manifest as performance or correctness problems. Examples are resource leaks prior to an application crashing, or incorrect use of synchronization locks prior to the application hanging. We do not handle bugs that lead to incorrect output, data corruption or failures that do not affect a system-measurable metric.

Design approach. We present ORION, a framework for localizing the origin of MM faults in distributed applications. ORION works by profiling a variety of metrics as the application is executing, either at declared instrumentation points (such as, method entry or exit) or asynchronously with a fixed periodicity. Through machine learning techniques, it verifies if the runtime profile is *similar enough* to profiles created offline of non-faulty application's executions. If it is not, ORION goes back through traces to indicate which metrics caused the divergence and from that, to the region of suspect code. The mechanism is probabilistic thus a rank-ordered list is provided to the developer for inspection. This design approach is shared with a few prior software systems [17], [18]. However, unlike these prior systems, which only gather traces from one or two dimensions of the application, e.g., CPU and memory, ORION performs application profiling along a large number of metrics. These metrics do not have to be hand-picked by the developer. ORION first automatically selects important metrics from

This work was partly supported by the National Science Foundation under Grant No. CNS-0916337, and it was performed partly under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344. (LLNL-PROC-632265)

the entire set for detailed analysis and then uses correlations between the metrics to diagnose subtle errors.

Summary of findings. We deploy ORION and evaluate it on diverse distributed applications: HBase [19], Hadoop [20], an on-campus Java Enterprise Edition (JEEE) application, called StationsStat, and an IBM regression testing application for its full system simulator called Mambo. We focus on failures that were difficult to debug manually. We demonstrate ORION with a total of 7 bug cases from these applications and find that the root cause is related to the top 3 abnormal metrics or code regions in 6 of them. The ORION code and data for this paper are available at [21].

The main contributions of this paper are:
(1) We introduce the concept of multi-dimensional metric profiling to provide problem determination for a wide variety of root causes. We show that, despite profiling a large number of metrics, it is possible to find the "needle in the haystack", i.e., the problematic metrics in a (possibly small) time window amidst a large number of normal metrics.
(2) We focus on a subtle class of problems that are not diagnosable by comparing instantaneous values of metrics against thresholds. Our approach develops a correlation-analysis algorithm that identifies when joint behavior of metrics is suspect.
(3) We successfully demonstrate ORION in four different distributed applications and under 7 failure conditions. In all of the cases, ORION pinpoints the origin of problems to abnormal metrics and in most of the cases it finds code regions that require human attention to fix the failures.

The rest of the paper is structured as follows. In Section II, we provide an overview of the execution of ORION. In Section III, we present the detailed design. In Section IV, we describe the seven case studies. In Section V, we review related work. In Section VI, we discuss practical implications of this work. Section VII concludes the paper.

II. OVERVIEW

A. Measurement Gathering

ORION collects measurements of multiple metrics at different levels in the system, i.e., hardware, OS, middleware and application by means of third-party monitoring tools. Using collected measurements, ORION builds models of normal behavior, which permits localizing the failures' origin.

The appendix shows the list of metrics that we measure. Although this is not an exhaustive list of all the metrics in the system, we tried to monitor as many metrics as possible from multiple layers. ORION does not impose a limit in the number of metrics used. Instead, it allows developers to use as many metrics as they think are useful for debugging. ORION addresses the *curse of dimensionality* problem by filtering out noisy metrics and automatically zooming into the metrics that are relevant for failure debugging.

Some examples of the metrics that we gather are:
End-user Application: per-servlet statistics such as processing time, request and exception counts; *Middleware:* cache hits and accesses, number of busy and created threads, and request processing time from the middleware layer (such as Apache Tomcat); *OS:* cpu- and memory-usage, context switches, file descriptors, disk reads/writes, packets received and transmitted, stack size; *Hardware:* L1/L2/L3 data and instruction cache hits and misses, TLB misses, branches taken/not taken/miss-predicted, load/store instructions, hardware interrupts.

Importance of collecting metrics from different layers. Bugs can manifest in different layers of the system, therefore it is necessary to monitor metrics from all layers. For example, a file-descriptor leak often manifests as an abnormal pattern in OS-related metrics such as memory and file-descriptor usage—Section IV-A illustrates this case in a file-descriptor leak bug in Hadoop. Other bugs can be diagnosed faster if hardware-related metrics are analyzed. To illustrate this, consider the sample code in Figure 1. Here, a performance problem would arise if a developer does not realize the importance of the sorting operation at line 8 and comments it out as a result. Then, due to random nature of the data, branch misprediction will drastically increase for the *if* statement at line 11 causing a performance degradation. In our experiments with multiple runs of this bug, performance can be reduced by a factor of 2.4x. To our astonishment, we found that many developers are not aware of such bugs—in fact this was one of the *most* voted topics in *stackoverflow* [22] and also one of the most viewed questions with more than 214,000 views. To further verify this fact, we conducted a survey with 30 respondents with 3 to 7 years of experience in software industry. Only 23% of them realized removing the sort statement might degrade performance.

In cases like in Figure 1, hardware metrics related to branch-misprediction would be more useful than other metrics in finding the problem's root cause. Since it is difficult for developers to select a priori the important metrics in

```
1  /*From a random data-set, add numbers greater than 127
       */
2  int arraySize = 50000;
3  int data[] = new int[arraySize];
4  Random rnd = new Random(0);
5  for (int c = 0; c < arraySize; ++c)
6     data[c] = rnd.nextInt() % 256;
7  /* No sorting causes branch misprediction */
8  //Arrays.sort(data);
9  long sum = 0;
10 for(int i = 0; i < 100000; ++i) {
11    for (int c = 0; c < arraySize; ++c)
12       if (data[c] >= 128)
13          sum += data[c];
14 }
```

Figure 1: Performance bug due to branch misprediction.

debugging, our tool starts with the entire set of metrics and automatically identifies those metrics with the zoom-in process that we describe in detail in Section II-C. For example, when we used our tool in the above scenario, it identified `branch-mispredicted` as the top anomalous metric. When we used only OS-level metrics, it was not able to accurately identify the root cause.

B. Profiling

ORION can perform multi-dimensional profiling in two ways: *synchronous* and *asynchronous*. In asynchronous mode, metric collection happens asynchronously to the application. The measurement gathering is done by a process separate from the application process(es). The asynchronous mode does not interfere with the monitored application and therefore is a lightweight method. ORION collects OS metrics values from the Linux `/proc` file system, hardware metrics through PAPI [23] and middleware- and application-metrics values from server containers by querying Java JMX connectors via a separate Linux process. This method requires offline processing to "line up" the metric collection points with the execution points of the application. This is done by using a common time base since all the involved processes execute on the same machine.

Synchronous profiling annotates code regions with a set of measurements. Whenever a code region begins and ends, this method collects measurements and labels them with the corresponding code-region name. For Java applications (such as in the Hadoop and HBase case studies), we use `Javaassist` to instrument binary code and to collect measurements at the beginning and at the end of classes/methods.

Notice that ORION can rely on other metric collection mechanism as long as they can provide at least the asynchronous mode of operation—the main novelty of ORION is its problem determination algorithms based on multi-metric data sets.

C. Workflow of our Approach

Figure 2 shows the steps in ORION to diagnose failures.
(1) Trace collection: ORION uses two set of traces to localize the origin of problems: a *normal* and an *abnormal* trace file. Normal trace files are obtained by collecting metrics of the application when failures are not manifested. This can be runs of an earlier bug-free application version or, in the case of intermittent failures, sections of a failed run where the fault did not manifest itself. The abnormal trace file is obtained when the failure is manifested. Labeling a run as one or the other is a manual process.
(2) Selection of pertinent metrics for modeling: ORION filters out unimportant (or noisy) metrics from the analysis. This step can also be viewed as dimensionality reduction. We used an algorithm based on Principal Component Analysis (PCA) to reduce the dimensionality of the problem. The intuition behind this step is to eliminate metrics that do not provide much information for the rest of the analysis; an example is constant metrics.
(3) Normal-behavior modeling: ORION creates a baseline model using the normal-behavior traces. Given traces of n metrics, the algorithm splits traces into equally sized time windows and calculates pairwise correlations between all the n metrics for each time window. These correlation serve as a summary of the expected behavior of the application in different time windows.
(4) Suspicious metric selection: Abnormal traces are used to select the metrics that are correlated with a failure. From all the n metrics, the top-3 most abnormal metrics are presented to the user. The user can then focus on finding the problem's origin based on the abnormal metrics.
(5) Abnormal code-region selection: Often, finding suspicious metric information in step 3 is not sufficient to infer the origin of a failure. For example, a metric like CPU utilization may be affected by any region of code. ORION selects the code regions that have a high degree of association with the suspect metric(s). ORION highlights suspicious code regions so that users can focus on finding bugs that could have caused the problems within them.

III. DESIGN

A. Selection of Pertinent Metrics for Modeling

Some dimensions (or metrics) are more important than others when debugging failures. We perform dimensionality reduction to: (i) eliminate redundant metrics from the analysis, and (ii) reduce computation overhead. We use Principal Component Analysis (PCA) as a baseline technique for dimensionality reduction. PCA uses an orthogonal transformation to convert a set of observations (of possibly correlated variables (into a set of values of linearly uncorrelated variables, i.e., principal components (PC). Applying PCA directly does not eliminate metrics from the analysis (which is our goal). Instead, PCA generates a new set of metrics which are linear combinations of the input metrics. We developed a heuristic following the idea in [24] to rank input metrics based on their importance. The idea is to rank metrics that are weighted heavily, especially in the first few PCs of the PCA-generated transformation (which contain most of the variance of the data), higher than metrics with smaller weights. Different from [24], we consider the contribution of each PC to explain the total variance in the data and design a new algorithm to rank metrics based on that.

To illustrate, consider Table I with 4 metrics **A**, **B**, **C**, **D** and 4 observation rows. In this experiment, the first principal component (PC-1) and the second principal component (PC-2) cover 46.7% and 45.3% of the variance of the data, respectively. In the transformation matrix generated by PCA, metric **A** has a weight of 0.7 in PC-1 and metric **D** has weight of 0.94 in PC-2. The heuristic used in [24] will

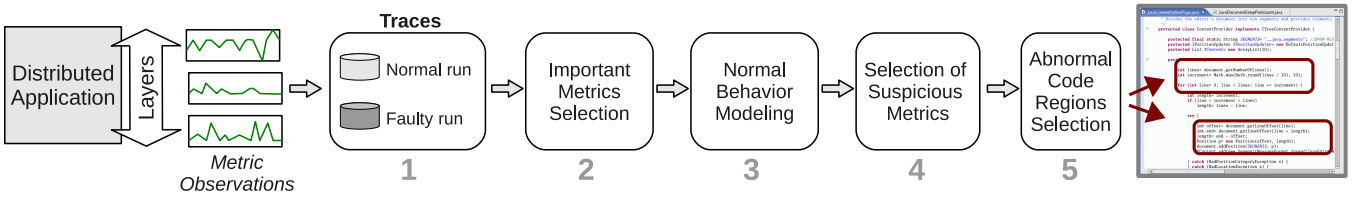

Figure 2: Overview of the problem determination workflow.

Table I: Almost equal variance explained by first two PCs.

Data				Transformation Matrix			
A	**B**	**C**	**D**	**PC-1**	**PC-2**	**PC-3**	**PC-4**
3	9	3	9	0.70	-0.11	0.68	-0.16
6	7	2	1	-0.21	0.31	0.46	0.80
1	6	2	1	0.66	0.07	-0.57	0.48
8	5	8	4	0.11	0.94	-0.03	-0.32
% of Variance Explained				**46.71**	**45.28**	**8.0**	**0**

```
1  metrics: array of metrics names
2  W: transformation matrix generated by PCA.
3
4  percentagePCA ← getVarianceExplainedByPCA(W,
       normalizedData)
5  for each column in W:
6    /* Multiply columns by variance coverage */
7    column ← column * percentagePCA[column]
8  for each row in W:
9    /* Calculate maximum value in row. Store in map */
10   rowMaxValueMap[row-num] ← MAX{abs(row)}
11 rank ← SIZE(metrics)
12 /*Sort based on max-weight in each row. Key is row-num
       */
13 for each key in valueSort(rowMaxValueMap)
14   rankMap[metrics[key]] ← rank
15   rank = rank - 1 /* Weight implies importance */
16 PRINT rankMap
```

Figure 3: PCA-based heuristic for filtering out metrics.

rank **A** higher than **D**. But if we also consider the variance covered by each PC, metric **D** should get a higher rank than metric **A**.

Figure 3 shows our ranking algorithm. Here, in the transformation-matrix, we first adjust the weight of a metric by the corresponding contribution of that PC towards explaining the total variance of the data. Then, we give equal importance to each column in the modified transformation matrix. We calculate maximum weight of a metric across all columns and then sort metrics based on these maximum weights to get a total ordering of metrics based on its importance. This allow us to filter out unimportant metrics from the rest of the analysis in ORION. The complexity of this algorithm is $O(n^2)$ where n is the number of metrics. We used the difference in ranks between normal and abnormal runs to chose important metrics. Performance bugs tend to change the correlation between metrics and in turn the weights in the PCA transformation matrix which changes relative ranking between metrics. More shift in metrics ranking indicates it was affected by the bug and hence should be chosen for detailed analysis. This first pass is very light weight and coarse as it lacks the notion of time. But it is very useful to reduce the overhead of detailed analysis in the following phase without losing vital information.

B. Modeling Sequential Data

Many bugs and performance anomalies develop a characteristic temporal pattern that can only be captured by analyzing measurements in a sequential manner (rather than by observing instantaneous snapshots of values). After reducing the number of metrics with the PCA algorithm, we build a baseline model that captures temporal patterns between metrics using correlation coefficients.

Observation window. Traces are split into non-overlapping windows of the same size. A window can be viewed as a matrix $S \times N$ in which S is the number of records (or samples) and N is the number of metrics. The set of records comprises one *observation window*. Since we do not know *a priori* the optimal size of observation windows (S), i.e., the window size that is sufficient to capture the temporal patterns that a failure shows, our algorithm sweeps through multiple sizes for the windows within a range (between sizes of 100 and 200 samples in our evaluation). The algorithm then finds the k-most abnormal windows (irrespective of its size) and, within those abnormal windows, the correlations and metrics that cause the unusual patterns. For our evaluation, we use a k value of 3. Section III-C describes our algorithms for the selection of suspicious metrics and code regions.

Correlation coefficients. For each observation window, ORION builds a vector of (pair-wise) correlation coefficients between all the metrics

$$CCV = [cc(1,2), cc(1,3), \ldots, cc(N-1,N))], \quad (1)$$

where $cc(i,j)$ is the correlation coefficient of metrics i and j, $i \neq j$. We denote this vector as a *correlation coefficient vector* or CCV. Correlation coefficients are calculated using the *Pearson correlation-coefficient* formula:

$$cc(X,Y) = \frac{1}{N-1}\sum_{k=1}^{N}\left(\frac{X_k - \bar{X}}{s_X}\right)\left(\frac{Y_k - \bar{Y}}{s_Y}\right) \quad (2)$$

where N is the number of elements of observations in the window, $\bar{X}$ and $\bar{Y}$ are the mean of variables X and Y

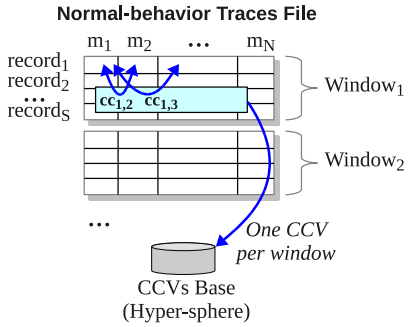

Figure 4: Creation of the normal-behavior hyper-sphere.

```
/* Get statistics of failed-run windows */
for each size s in range r:
  setOfWindows ← create windows set of size s
  for each window w in setOfWindows:
    d ← find NN distance of w in the hyperSphere
    ccs ← get top abnormal corr. coefficients of w
    Append {w, d, ccs} to tuplesList

/* Select the most abnormal windows*/
Sort tuplesList based on distance d (high to low)
abnormalWindows ← get top elements in tuplesList

/* Build histogram of most abnormal metrics */
for each window w in abnormalWindows:
  for each correlation cc corresponding to w:
    From cc add metrics X and Y to histogram

Print the most frequent metrics in histogram
```

Figure 5: Algorithm to select the suspicious metrics from traces of a failed run.

respectively, and s_X and s_Y are the standard deviations of X and Y.

Normal-behavior Model. Using the normal-behavior traces, our framework creates a baseline model which is used in step 3 (from the main workflow) to select suspicious metrics. The model is a set of normal-behavior CCVs obtained by splitting traces into observations windows and computing a CCV for each window. We term this model as a *hyper-sphere*. Figure 4 shows the process of creating this hyper-sphere. The number of points in it corresponds to the number of windows that we obtained from the normal-behavior traces, and the dimensions (or features) are correlation-coefficients of metric pairs. Notice that, if we have N metrics in the analysis, the dimension of the hyper-sphere is $D = \frac{N(N-1)}{2}$.

The idea of using a hyper-sphere where dimensions are correlation coefficients is that we can use nearest-neighbor to pinpoint abnormal observation windows from the faulty traces. Since an observation window is translated to a single data point (i.e., a CCV), we can treat the problem of finding abnormal windows as an outlier detection problem via nearest-neighbor, i.e., an abnormal window would correspond to the CCV point that is the farthest away from the hyper-sphere.

C. Detection of Suspicious Metrics

Motivation. The main motivation of our technique is that the manifestation of faults will change the correlations between some (affected) metric(s) and the rest of the metrics, while maintaining the legitimate correlations in the other metrics. To illustrate this idea, consider a bug where unused database connections are kept open—metrics such as file descriptors and open sockets will be affected by the bug and will exhibit a different temporal pattern than during workloads where the bug is not activated. However, correlations among the other metrics will not be affected.

Our goal is that, when faults are manifested, ORION finds the metric(s) that is (are) mostly associated with failures. This is performed by ranking metrics according to their contributions to correlation breakups and by selecting the top-k metrics in this ranking. The application developer can subsequently focus on reviewing the code which affects these suspicious metrics to locate the root cause of the problem.

Algorithm overview. The goal of the algorithm is to select the metrics that are most likely associated with the problem's origin. The algorithm's input is a normal-behavior hyper-sphere and traces of a failed run. The algorithm's output is a list of metrics that are ranked by abnormality degree. The algorithm is presented in Figure 5.

The algorithm has the following main steps:

Statistics creation per window: We create observation windows of multiple sizes from the failed-run traces file. For each window, we calculate two statistics: (1) the *nearest-neighbor* (NN) distance of the window from the hyper-sphere representing normality. This distance is calculated by first computing a CCV from the window and then by finding the euclidean distance between the CCV and the closest point in the hyper-sphere using the formula $d = \sqrt{\sum_{i=1}^{D}(cc_i - bb_i)^2}$, where cc_i and bb_i are correlation coefficients; (2) the dimensions that have the highest weight in making the CCV far away from the hyper-sphere. A dimension here corresponds to a correlation coefficient.

Abnormal window selection: Windows are sorted by their NN distance from high to low and only the top-k windows in the list are taken for further analysis ($k = 3$ for our evaluation). These windows correspond to time periods when abnormal behavior is manifested.

Select most abnormal metrics: Once the top-k abnormal windows are ranked, within each window, the correlation coefficients (CCs) are ranked by how much they contribute to the NN distance of that window. Now the top-k CCs are taken from each abnormal window, giving a total of $k \times k$ CCs. Recall that each CC involves two metrics. With these short-listed CCs, the metrics that are present in them are

```
/* Get statistics of failed-run windows */
for each size s in range r:
  normalWins ← get windows from normal traces
  failedRunWins ← get windows from abnormal traces
  for each window w in failedRunWins:
    d ← NN distance of w from normalWins
    Append {w, d} to tuplesList

/* Select the most abnormal windows*/
Sort tuplesList based on distance d (high to low)
abnormalWindows ← get top elements in tuplesList

/* Build histogram of abnormal code-regions */
for each window w in abnormalWindows:
  Add code regions in w to histogram

Print the most frequent code regions in histogram
```

Figure 6: Algorithm to select the suspicious code regions from traces of the failed run.

counted up and the top-k most frequently occurring metrics are flagged as the most abnormal metrics. Notice that we use the same parameter for filtering the top choices (windows, CCs, metrics). In theory, they are different parameters, but in practice the same value ($k = 3$) works well and reduces the search space of parameters, a desirable outcome for any deployable tool.

D. Detection of Anomalous Code Regions

After the suspicious metrics are detected, ORION highlights code regions that make metrics abnormal so that developers can focus on them to fix the problem. ORION first finds suspicious periods of time in which a metric shows an unusual temporal pattern (i.e., an abnormal window). Then, within that period, ORION looks for outlier observations, i.e., an abnormal code region.

Algorithm requirements. The algorithm for detecting abnormal code regions is similar to the abnormal metric-selection algorithm. A major difference is that only one metric is used in the analysis, i.e., the abnormal metric. This metric is given by the previous step, i.e., the metric-selection step. The user can opt to execute this algorithm using the top-two (and top-three and so on) abnormal metric(s) if the top-one abnormal metric does not help in finding the problem's origin. The algorithm's inputs are traces files (from the normal and the failed run) such as in Figure 4 but with only one column—this column corresponds to measurements of the abnormal metric. The output of the algorithm is the top few anomalous code regions. We assume that each record in this file is annotated with a code region.

Algorithm overview. The algorithm is shown in Figure 6. First, we construct a set of windows from traces of the normal run and another set from traces of the failed run. Second, we find NN distances of the windows of the failed-run from the normal-behavior windows. Then, to select the most abnormal windows, we rank the failed-run windows based on the NN distances (from high to low) and select only the top-k windows. Finally, we build a histogram of the occurrences of code regions in these abnormal windows—as we observe from our case studies, the faulty code regions in performance bugs execute frequently in the most unusual periods of time. The top-3 most frequent code regions are shown to the user.

How to compare one-dimensional windows? In the previous algorithm, we find the difference between two windows by calculating the Euclidean distance of their corresponding $CCVs$. We summarize the window's information by calculating aggregates of its values: *average*, *standard deviation*, *minimum*, *maximum* and *sum*. These aggregates become the features of a window. ORION then finds the dissimilarity between two windows by computing the Euclidean distance using these aggregates as features.

IV. EVALUATION

In this section, we describe how we debugged seven performance bugs in different distributed applications: Hadoop, HBase, a distributed system heavily used in IBM as regression framework, and a distributed application used by students in a large university. Due to space limitations, we only present in detail the first 4 cases and provide a summary of the results of cases 5–7. In all of the cases, ORION reduces substantially the time spent in localizing the problem origin by showing the metric most perturbed by the fault, and if needed further, the abnormal code region. The process is fully automated so users do not need to have full understanding of the application and its components dependencies.

A. Case 1: Hadoop DFS

Hadoop is an open-source framework that supports data-intensive distributed applications [20]. It enables applications to work with thousands of computational nodes and a large amount of data. We use ORION to diagnose a file descriptor-leak bug that occurred in the Hadoop Distributed File System (HDFS) in version 0.17. The bug report is HADOOP-3067.

We collect all the OS and hardware level metrics given in Table IV via synchronous profiling. All the Java classes and public methods within each class are instrumented. Since we are debugging the Hadoop DFS, we only consider the `java/org/apache/hadoop/dfs` package. A total of 45 Java classes and 358 methods in these classes are instrumented.

This bug is manifested as a failure in one of the HDFS tests (the `TestCrcCorruption` test.) The bug origin is that subclasses `DFSInputStream` and `DFSOutputStream` of the main class `DFSClient` did not handle open sockets correctly by not closing them when they are not used anymore. The patch that developers suggested to fix the bug included changes

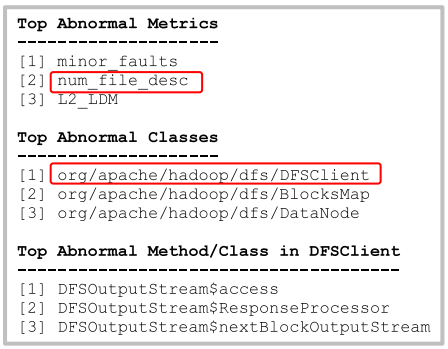

Figure 7: Results from ORION for the HDFS bug.

Table II: Average use of file descriptors per class in HDFS for the specific bug discussed in Section IV-A.

Rank	Class	Average # File Descriptors
1	NamespaceInfo	6.0
2	INodeDirectory	1.31
3	INode	1.29
4	UnderReplicatedBlocks	1.25
5	DatanodeInfo	1.24
6	DataNode	1.21
7	DatanodeBlockInfo	1.2
8	DFSClient	1.16
9	DataBlockScanner	1.14
10	NameNode	1.13

to the following code: class `DFSClient`, subclasses `DFSInputStream` and `BlockReader` (which is used internally by `DFSOutputStream`). We used the buggy version and revision 0.17 to obtain traces of a failed run, and code from a previous revision where the `TestCrcCorruption` test passed to get traces of a normal run.

Figure 7 shows ORION's results. The top-three abnormal metrics presented by ORION are: (1) `minor_faults` (No. minor page faults), (2) `num_file_desc` (No. open file descriptors), (3) `L2_LDM` (level-2 load miss). The 2nd metric is associated with the problem's origin since an increase in the number of open sockets caused by the bug affects directly the number of open file descriptors. Metrics (1) and (3) are both memory related and they are pinpointed as suspect because the bug also causes abnormal memory consumption patterns.

ORION also presents abnormal code regions, first, based on Java classes and second, based on subclasses (of the abnormal classes) and methods within them. ORION correctly pinpoints `DFClient` as the most abnormal class. Within `DFClient`, ORION highlights `DFSOutputStream` as the main abnormal subclass. This is only partially correct—part of the bug fix is in `BlockReader` which is used internally by `DFSOutputStream` and `DFSInputStream`; however, `DFSOutputStream` does not require changes to fix the bug.

To see if a simpler, and currently practiced, approach can lead the developer to the origin of the bug faster, we set up the following hypothetical steps for hunting this bug.

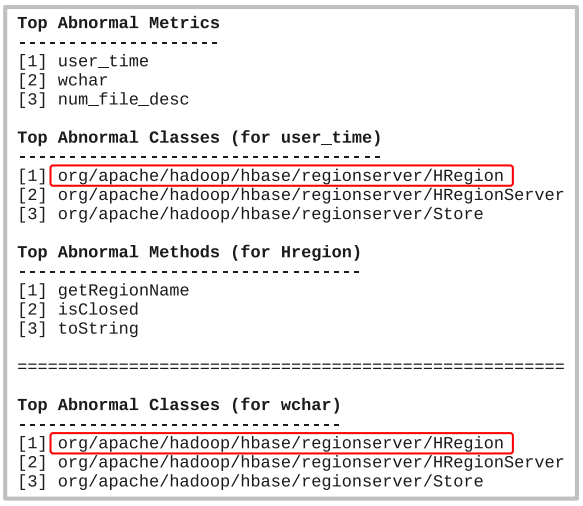

Figure 8: Results from ORION for the HBase bug.

Suppose that a simple profiling tool indicates a high number of file descriptors in use. The developer then proceeds to examine which classes use file descriptors most. The answer to this question is shown in Table II. The average number of file descriptors used per method is calculated by taking an average across all invocations of the methods of that class. From this, the developer would be likely to inspect the classes appearing near the top. It is only when one gets to the 8th ranked class that one gets to the class where the bug lies, `DFSClient`. Thus, this will lead to significant time manually inspecting classes 1–7 and ruling them out as the source of the bug.

B. Case 2: HBase

HBase is an open-source, distributed, column-oriented database [19]. It operates on top of distributed file systems like the HDFS and is capable of processing very large scale of data with MapReduce. We use ORION to collect and analyze the metrics of a deadlock bug in HBase 0.20.3 (bug report HBASE-2097). We collect all the OS-level metrics shown in Table IV. There are 27 java classes that are instrumented from the `hadoop/hbase/regionserver/` package. They include classes to handle region columns, store data files, logs and many other abstractions. These classes include 184 methods which are all instrumented.

The bug, which manifests as an application's hang, is the result of two locks being acquired in an incorrect order. The bug lies in two methods, `HRegion.put` and `HRegion.close`. It is activated by running the HBase `PerformanceEvaluation` testing tool which is used to evaluate HBase's performance and its scalability. The bug manifestation is intermittent—it manifests on average 75% of the time—making it particularly difficult to localize.

We ran a previous bug-free version to generate normal-behavior traces and applied ORION against the traces of a

failed run when the deadlock manifests. Figure 8 shows the results. The top abnormal metric, i.e., `user_time`, is the amount of user-level CPU time. This metric *per se* does not provide much insight into the failure origin since it is difficult to correlate that to a code region. We observe that, using `user_time` as our abnormal metric, the most abnormal code region is `HRegion` class, which is where the bug lies. Further, considering the 2nd most abnormal metric `wchar` (the number of bytes which the program has caused, or will cause to be written to disk), the flagged code region is also the `HRegion` class. This confirms that `HRegion` is where the developer needs to focus her attention.

The bug patch shows that the bug resides in the `HRegion` class. This class stores data for a certain table region and all columns for each row—a given table consists of one or more HRegions. The patch flips the order of acquiring the two locks (a write lock and then a read lock) and consequently the order of releasing them. It puts the change in both the `HRegion.put` and the `HRegion.close` methods. We speculate that spinning on locks in the deadlock situation causes the `user_time` metric to go awry.

The abnormal methods within `HRegion` that are flagged by ORION are `getRegionName`, `isClosed` and `toString` (Figure 8). They do not correspond to the methods where the bug lies (`put` and `close`). Through a detailed investigation, we identify the cause of this. The three flagged methods are invoked much more often within `HRegion` than are the erroneous methods. However, the three flagged methods and the erroneous methods occur close together in time. The algorithm in ORION, after it has zoomed into a time window where the fault manifested itself, considers frequencies of methods within that suspect time window to decide which methods to flag. This causes it to flag the most frequently invoked `getRegionName`, `isClosed` and `toString` methods.

C. Case 3: StationsStat

StationsStat is a Java multi-tier application that is used to check the availability of workstations on Purdue's computing labs. Students on the campus use StationsStat daily to check the number of available Windows or Mac workstations for each lab on campus. StationsStat is managed by Purdue's IT department (ITaP) and runs in Apache Tomcat 5.5 on RedHat Enterprise Linux 5 with an in-memory SQL DB.

Due to an unknown bug, periodic failures were observed in which the application became unresponsive. System administrators received failure reports from their monitoring system, Nagios, or from user phone calls. Since the problem root cause was unknown, the application was restarted and the problem appeared to go away temporarily. StationsStat's administrators tracked 495 metrics from the OS, middleware, and application layers at 1 minute intervals (using asynchronous profiling) for more than two months. We collect the application, OS and middleware metrics in Table IV.

```
Top Abnormal Metrics
--------------------
[1] Servlet:AxisServlet-WebModule/processingTime
[2] Javax.sql.DataSource/infdbd2/numActive
[3] rss
```

Figure 9: Results from ORION for the StationsStat case.

StationsStat was a challenging scenario, not only because the problem's root-cause was unknown, but also because there was no error-free data available to create the normal-behavior hyper-sphere. Fortunately, ORION can still work in this scenario by using *almost error-free* data. The administrators noticed that, after restarting StationsStat, the next failure was often seen only after a week or more—symptoms seemed to suggest that the problem, possibly a resource exhaustion bug, grew progressively from a service restart to a failure. ORION therefore used a data segment collected right after a restart to build the hyper-sphere representing normality. ORION also filtered out constant metrics in this phase which resulted in 70 non-constant metrics for the rest of the analysis.

In contrast with the previous cases, we conducted a blind experiment in which ORION gives us the suspicious metrics without us knowing the actual root-cause of the problem. Figure 9 shows the abnormal metrics that ORION finds. We then compared ORION's answer to the application developers best guess of the root-cause. The results show that the suspicious metrics given by ORION matched well with what the developer thought to be the origin of the problem. The 2nd abnormal metric is the number of active SQL connections. The application had only one localized region where it made calls to the SQL driver that Tomcat used to handle database connections. The developer concluded that the SQL driver code was buggy since it was obvious that the few lines of SQL driver invocation code in his application was not. Upgrading the SQL driver fixed the problem and the application continues to run today providing an important function to students all over campus. Interestingly, the top metric flagged by ORION—the processing time of a servlet— had nothing to do with the bug. We found that this is due to large differences in workload between our normal and abnormal data sets (normal data were collected right after the server restart, while abnormal data were collected after the server has been up for a while). This negative result highlights the importance of getting the normal and the abnormal data sets under similar workload conditions.

In this case study, there was no need for the additional step of ORION where it maps the abnormal metric to a code region. This is because only one small localized region of the application code had anything to do with SQL, which was implicated by the metric.

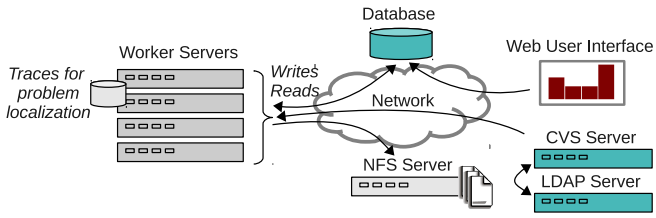

Figure 10: Mambo Health Monitoring system.

D. Case 4: Mambo Health Monitor

The Mambo Health Monitor (MHM) is a regression test system for the IBM Full System Simulator, commonly known as Mambo. Mambo is a computer architecture simulator for systems based on IBM's Power(TM) architecture. Mambo has been used in the development and testing of a wide range of systems, including IBM's Power line of server systems (Power5, Power6, Power7), the Cell(TM) processor used in the Sony Playstation3(TM), and IBM's BlueGene systems. The MHM executes tests on the simulator to detect regressions in behavior that may be introduced by new development. The tests are drawn from a test suite that covers the key functionality in all the major target systems. Test results are stored in an SQL database and are accessed through a web-based interface. Figure 10 shows the system's elements.

There is a servers farm which serve as MHM clients. Each client accesses a database to determine which tests have to be run. The client then checks out the code from a CVS repository (after authentication) and proceeds to run the test. The test execution sometimes requires specialized resources from the node on which it is executing, such as, a virtual network port. Upon completion, the client writes results in the database (success or failure) along with some informational items, such as, performance results.

Failures. A test-case can fail due to a problem in the environment or a problem with the architecture being simulated. Examples of environment-related problems causing a test to fail are many: a NFS connection that fails intermittently, a cron job fails to get authenticated with the LDAP server, Linux failing to map the simulator's network port to the machine's network port, /tmp filled up. A problem like this can make a developer falsely believe that her architecture code is buggy when in reality the problem lies in the environment. A key source of difficulty is that these problems are often transient and the software elements do not have error messages that correspond to the actual problem. We choose the problem of losing NFS mount as it has been a frequent problem for users over its seven year lifespan. We emulate NFS problems by dropping outgoing NFS packets with a probability of 0.1. The NFS packet dropping functionality is implemented by adding an iptables rule at the start of the faulty run.

```
Top Abnormal Metrics
--------------------
[1] wchar
[2] read_bytes
[3] rss

Top Abnormal Code Regions (for wchar)
-------------------------------------
[1] Code to avoid problems with X tests
[1] Code to avoid running tests on debug builds
[1] Checking for more commits we could work on
[1] Disconnecting from the database
```

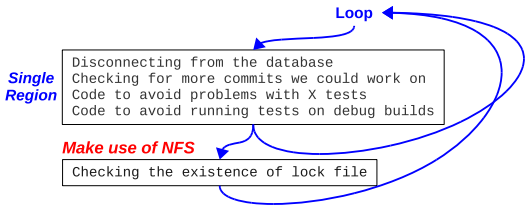

Figure 11: Results from ORION for the MHM problem.

Code annotations. We run ORION in an asynchronous mode. The profiling process collects metrics at a 1 sec granularity. MHM was instrumented at 48 instrumentation points in 1400 lines of Tcl source code, which resulted in 2227 records. The Tcl script invokes Perl and bash scripts. Since these had been rigorously tested, we were told that they should be kept out of scope of our problem localization effort. The instrumentation code records a timestamp and an identifier for the code region. Unlike in the other applications, this code did not have finely granular methods (and of course no classes). Therefore, the points to insert instrumentation was a subjective decision and this was done based on the amount of comments in the code. Our instrumentation covers the starts and the ends of crucial operations, such as, CVS operations, NFS operations, and database operations, also other structures, such as `if-then-else` blocks.

Results. We collect traces of a normal and of a failed run. We use the same machine as the MHM client and keep the workload pattern the same. Figure 11 shows the results of applying ORION. First, notice that the two top abnormal metrics are metrics related to I/O, i.e., `wchar` and `read_bytes` (written characters to disk and read bytes from disk, respectively). Next, we find abnormal code regions using `wchar` as our abnormal metric—ORION ranks equally four different code regions as the figure shows. Notice that none of the pinpointed regions perform any operation that makes use of NFS, the root cause. However, when we look at the code (Figure 11), we notice that these regions are short and are always executed together inside a loop, so they can be grouped into one region. We also notice that, right after this (grouped) region, there is a code region that makes use of the NFS, i.e., the `Checking-the-existence-of-lock-file` region. This region performs I/O to access a file that is mounted using NFS and is affected by the injected fault.

The reason the NFS region is not ranked as an abnormal

region (in fact is ranked 4) is that measurement inaccuracies emerge from asynchronous profiling. These code regions (demarcated by our instrumentation points) are small compared to the frequency of metric collection. Hence, it becomes difficult to accurately map the metric collection points to within an instrumented code region. In these cases, ORION pinpoints to the user not only the abnormal code region but also one region before and one region after the abnormal one. Hence, a design decision in ORION is that when asynchronous profiling is used and the instrumented code region is "small", ORION pinpoints to the user not only the abnormal code region but also one region before and one region after the abnormal one. This strategy works well here since the code region that is affected by the fault is right after the region that ORION selects as abnormal.

E. Summary of Cases 5–7

Case 5. This bug (HBASE-3098 report) manifests itself as an application hang in HBase version 0.90. ORION results suggest using the `vsize` metric as the abnormal metric to find the abnormal code regions. ORION ranks the following classes as the top-3 most abnormal: `HRegionServer`, `ZooKeeperWatcher`, `HBaseServer`. The fixing patch for this bug (suggested by the developers) contains changes to 12 Java classes, one of this being `HRegionServer`. We noticed that the the `Hmaster` class, which is ranked as number 4 in ORION, is also part of the patch. Two buggy code regions showed up in top 5 results. There are a total of 131 classes that the developer would need to analyze to fix this bug. ORION reduces the search space substantially to a handful of classes.

Case 6. This bug (HBASE-6305 report) manifests itself as a hang in the `TestLocalHBaseCluster` test in Hbase version 0.94.3. According to the bug report, the origin of the fault appears to be a stack overflow error. Using ORION we find the following top-3 abnormal code regions: `HBaseServer`, `HRegionServer`, `ServerName`. The suggested patch fixes only two classes—one of those is `HRegionServer` which was ranked by ORION as the second most abnormal class. There is a total 155 classes that the developer would need to analyze in order to fix this bug without ORION.

Case 7. This bug (HBASE-7578) was one of the most difficult to debug since it only manifested occasionally (in the `TestCatalogTracker` test case) as a hang. The top-3 abnormal classes given by ORION did not contain the origin of the bug, i.e., they were not changed by the suggested patch. However, ORION was able to show `CatalogTracker` which was part of the patch, in its top-5 results. Also notice that the developer would have had to analyze more than 44 classes to fix this bug without the use of ORION.

Table III: Performance measurements.

Case No.	Bug-report ID	Bug localization time (min)
1	Hadoop-3067	4.06
2	HBase-2097	15.32
3	StationsStat	31.04
4	JHM	0.48
5	HBase-3098	4.90
6	HBase-6305	10.35
7	HBase-7578	2.10

F. Performance

We measure the time ORION spends in *bug localization*. This involves creating the hyper-sphere based on normal-behavior traces, finding the abnormal metric and, subsequently, the abnormal code region. Table III shows our measurements. The localization time depends on the amount of traces and metrics that are collected in a bug case. The amount of traces depends on the number of instrumented functions and their call frequency.

V. RELATED WORK

Various tools exist to help programmers and network administrators localize the problems in distributed applications. Most of these tools can be grouped into these main categories: traditional debuggers and execution replays, model checkers, and statistical methods. Each type of tools is appropriate for different scenarios, as we will discuss below.

Traditional debuggers such as gdb, and tools that enable execution replays [8], [9] let the programmer find the exact line of code of the bug. However, for this to be feasible, the programmer needs to have a good guess of where the problem lies. Therefore, these tools are complimentary to our tool since our tool reports suspicious regions of code, and when it is not obvious where the bug is, the programmer can use these tools to pinpoint the bug. Attariyan *et al*. [3] propose a technique that determines the root cause of performance problems using taint analysis. It is suitable for cases where the source of the problem is user input or the configuration file.

Model checkers are useful for checking small applications against specifications [10], [11]—due to its exhaustive nature, it is not feasible for most real-world applications. In addition, it is more appropriate for checking against specifications. Liu *et al*. [12] propose a technique that does live model checking and provides execution replay. The programmer writes a predicate that is invariant throughout the execution, and this predicate is checked as the application runs. When the predicate is violated, the system states leading to the violating state are given as output. While this approach works well for specifications, as the instruction that changes the system state from conforming to violation is usually the root cause of the problem, for other types of problems such as performance problems, the errors may have accumulated from different regions of the code before

the specification is violated. Our tool does not assume that the instruction that makes the system state in violation is the root cause of the problem, and is thus more applicable to a wider range of problems.

There is a volume of work on statistical methods to detect and localize problems. Some of the work analyzes application logs [13], [14], [15], [16]. However, there is often a one-to-many mapping between the log record corresponding to the problem and the actual code regions that could be the source of the problem. [25], [26], [27] analyze request flows to diagnose problems in request-processing applications. Other work analyzes metric values, typically using machine learning algorithms [17], [28], [29], [30]. In [17] and [28], the signature of the current problem is compared to a database of known problems. If there is a match, the diagnosis and fixes used previously can be reused again. This approach is suitable for problems that are not easy to fix even if the root cause is known (e.g., overloaded servers), problems due to the environment, or hardware problems. In other situations, once a problem is diagnosed and fixed, it will not occur again, limiting the usability of the tool. The overall approach of [29] and [30] is similar to our approach in that machine learning models are trained based on training data. If the system is in an abnormal state, the metrics that are most abnormal are given to localize the problem. In addition, when this is not enough to pinpoint the location of the fault, ORION goes a step further and provides a ranking of most suspicious code regions to reduce the programmer's effort needed to fix the problem.

VI. DISCUSSION

We discuss practical limitations of this work:

- We observe that, as ORION drills down deeper looking for problematic code regions (class→subclasses→method), it provides less accurate results. An example is the Hadoop bug in which we are able to identify the abnormal code region at a Java class granularity but not at a method granularity.
- Applications that do not provide code-region delimiters would require manual effort from the developer (like with the MHM system) to indicate what are good instrumentation points. For most applications, however, ORION automatically annotates entry and exit points of methods.
- A trade-off of our asynchronous profiling approach is the difficulty of mapping metric samples to code regions accurately, when the code region is short relative to the time it takes to sample metrics. However, this comes with the advantage of minimal perturbation of the application. For asynchronous profiling, ORION provides code regions that are adjacent to whatever code region it finds as abnormal. For example, if the bug arises only when a particular package is updated, she could only instrument that package to look for the bug.
- Although in bug cases 1, 2 ,5–7 we collect metrics from a single node (due to the nature of the test cases), our approach also works for multi-node metrics. In such a case, ORION would have to label metrics from different nodes and treat them as different metrics (or features). Cases 3 and 4 are fully distributed and do not need this special treatment.

VII. CONCLUSION

We propose ORION to perform root cause analysis for failures in distributed applications. From a comprehensive set of monitored metrics, ORION pinpoints the metric and a window that is most highly affected by a failure and subsequently highlights the code region that is associated with the problem's origin. Our algorithm models the application behavior through pairwise correlations of multiple metrics, and when failure occurs, it finds the correlations (and hence the metrics) that deviate from normality. Our case studies with different distributed applications show the utility of the tool—ORION can localize the origin of real-world failures at a granularity of metrics and code regions in few minutes.

REFERENCES

[1] "The GNU Project Debugger," http://www.gnu.org/software/gdb/.

[2] "Using JConsole to Monitor Applications," http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html.

[3] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proc. OSDI*, 2012.

[4] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[5] ".NET Memory Profiler," http://memprofiler.com/.

[6] "JProfiler," http://www.ej-technologies.com.

[7] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," in *Proc. NSDI*, 2007, pp. 20–20.

[8] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: global comprehension for distributed replay," in *NSDI*, 2007.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *USENIX ATC*, 2006.

[10] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: finding liveness bugs in systems code," in *Proc. NSDI*, 2007, pp. 18–18.

[11] M. S. Musuvathi, D. Park, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," in *Proc. OSDI*, 2002.

[12] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3s: debugging deployed distributed systems," in *Proc. NSDI*, 2008, pp. 423–437.

[13] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. ICDM*, 2009, pp. 149–158.

[14] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. NSDI*, 2012, pp. 26–26.

[15] S. Sabato, E. Yom-Tov, A. Tsherniak, and S. Rosset, "Analyzing system logs: a new view of what's important," in *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, ser. Proc. SYSML'07, 2007, pp. 6:1–6:7.

[16] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. SOSP*, 2009, pp. 117–132.

[17] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proc. EuroSys*, 2010, pp. 111–124.

[18] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *SC*, 2006.

[19] "Apache HBase," http://hbase.apache.org/.

[20] "Apache Hadoop Project," http://hadoop.apache.org/.

[21] "Git repos of data and source code," https://github.com/ilagunap/Orion, https://github.com/subrata4096/.

[22] "Branch mispredition for unsorted array," http://stackoverflow.com/questions/11227809/.

[23] "PAPI," http://icl.cs.utk.edu/PAPI/.

[24] K. Bare, S. Kavulya, and P. Narasimhan, "Hardware performance counter-based problem diagnosis for e-commerce systems," in *Proc. NOMS*, 2010, pp. 551–558.

[25] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004.

[26] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Proc. NSDI*, 2004, pp. 23–23.

[27] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proc. NSDI*, 2011, pp. 4–4.

[28] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. SOSP*, 2005, pp. 105–118.

[29] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proc. ICDCS*, 2009, pp. 623–630.

[30] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *DSN*, 2005, pp. 644–653.

[31] "Metrics table," https://engineering.purdue.edu/dcsl/orion.

APPENDIX

Due to space limitations, Table IV only shows some of the metric types that we analyzed, grouped by layer. The full list of metrics and descriptions can be seen here [31]. Notice that some metrics have multiple instances. For example, the `data_source_numActive` is the number of active connections per database—the StationsStat system has two databases so it has two instances of this metric. Similarly, Java servlets, containers and server metrics have several instances.

Table IV: Some of the used metrics.

Hardware Metrics	**OS Metrics**
L1_DCM	minor_faults
L2_DCM	major_faults
L2_ICM	user_cpu_time
L1_TCM	sys_cpu_time
L2_TCM	num_threads
CA_SHR	virt_mem_size
CA_CLN	rss_mem_size
CA_ITV	processor
TLB_DM	stack_size
TLB_IM	read_chars
L1_LDM	write_chars
L1_STM	read_bytes
L2_LDM	write_bytes
L2_STM	canceled_write_bytes
HW_INT	num_file_desc
BR_CN	nicRcvBytes
BR_TKN	nicRcvPckts
BR_NTK	nicSentBytes
BR_MSP	nicSentPckts
BR_PRC	IPInTruncatedPkts
TOT_IIS	IPInOctets
TOT_INS	IPOutOctets
VEC_DP	**Application Metrics**
FP_INS	servlet_processingTime
LD_INS	servlet_maxTime
SR_INS	servlet_requestCount
BR_INS	servlet_errorCount
VEC_INS	datasource_maxWait
RES_STL	datasource_numIdle
TOT_CYC	datasource_maxActive
L1_DCH	datasource_numActive
VEC_SP	**Middleware Metrics**
L1_DCA	request_handler_bytesSent
L2_DCA	request_handler_bytesReceived
L2_DCR	request_handler_requestCount
L2_DCW	request_handler_maxTime
L1_ICH	request_handler_processingTime
L2_ICH	request_handler_errorCount
L1_ICA	cache_hits
L2_ICA	cache_accesses
L2_TCH	number_threads
L1_TCA	
L2_TCA	
L2_TCR	
L2_TCW	
FML_INS	
FDV_INS	
FP_OPS	
SP_OPS	
DP_OPS	